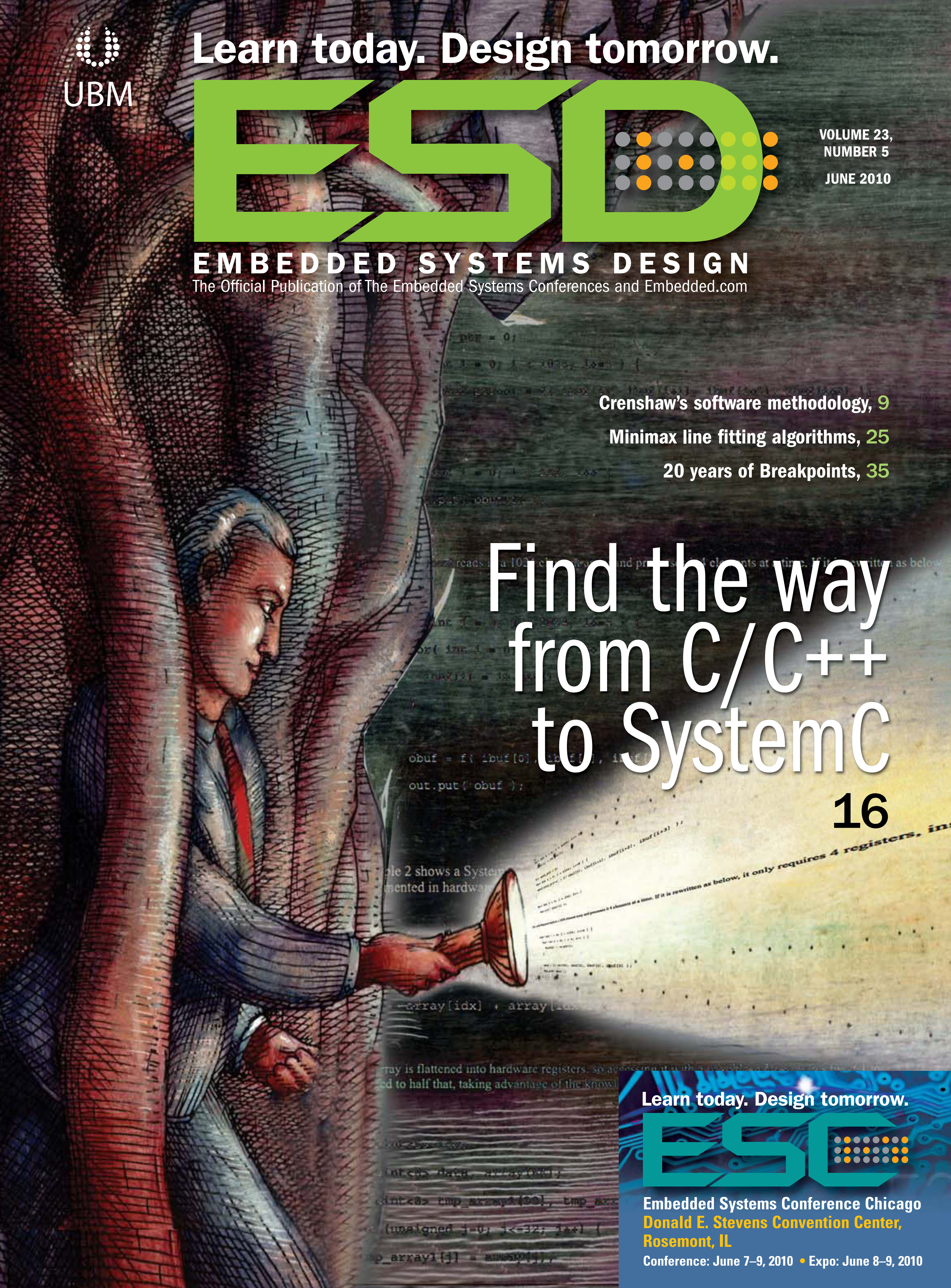


UBM

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

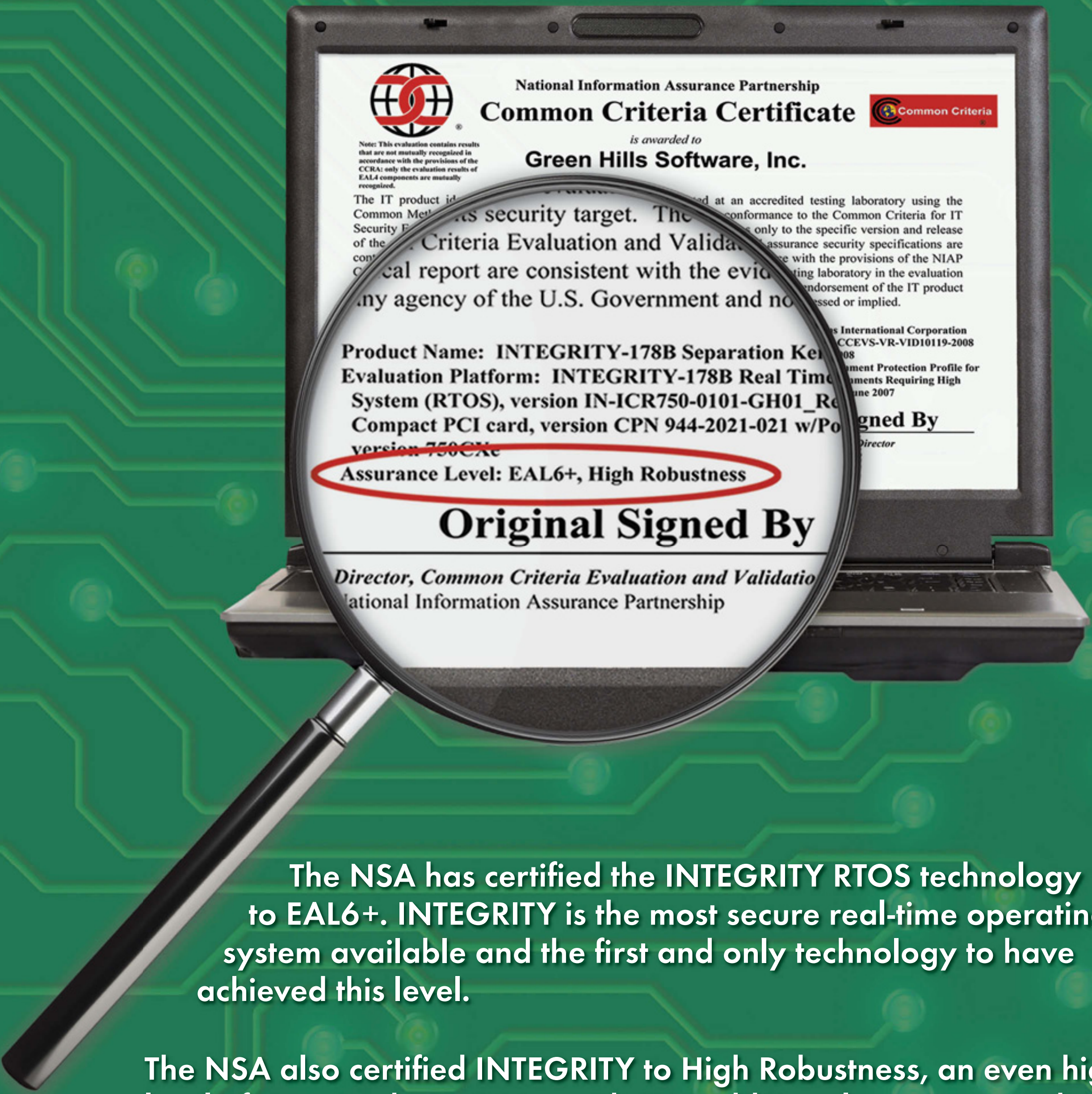
The Official Publication of The Embedded Systems Conferences and Embedded.com

VOLUME 23,
NUMBER 5

JUNE 2010

CDC-ACM
USB 3.0
OHCI
RFI
HOST
Composite Device
Compos
Device
ULPI
EHCI
Mass Storage
USB 2.0
MTP
HID
USB 1.1
O

INTEGRITY RTOS has it. No one else does.



National Information Assurance Partnership

Common Criteria Certificate

is awarded to

Green Hills Software, Inc.

Note: This evaluation contains results that are not mutually recognized in accordance with the provisions of the CCRA: only the evaluation results of EAL4 components are mutually recognized.

The IT product id… …ed at an accredited testing laboratory using the Common Met… …ts security target. The… …conformance to the Common Criteria for IT Security E… …only to the specific version and release of the… Criteria Evaluation and Valida… …assurance security specifications are con… …e with the provisions of the NIAP …cal report are consistent with the evid… …ting laboratory in the evaluation …ny agency of the U.S. Government and no… …ndorsement of the IT product …essed or implied.

…s International Corporation
…CCEVS-VR-VID10119-2008
…08

…ment Protection Profile for …ments Requiring High …une 2007

Product Name: INTEGRITY-178B Separation Ke…

Evaluation Platform: INTEGRITY-178B Real Tim… System (RTOS), version IN-ICR750-0101-GH01_R… Compact PCI card, version CPN 944-2021-021 w/Po… version 750CXe

Assurance Level: EAL6+, High Robustness

Original Signed By

Director, Common Criteria Evaluation and Validatio…

…ational Information Assurance Partnership

The NSA has certified the INTEGRITY RTOS technology to EAL6+. INTEGRITY is the most secure real-time operating system available and the first and only technology to have achieved this level.

The NSA also certified INTEGRITY to High Robustness, an even higher level of security than EAL6+, with 133 additional security mandates over and above the 161 required for EAL6+.

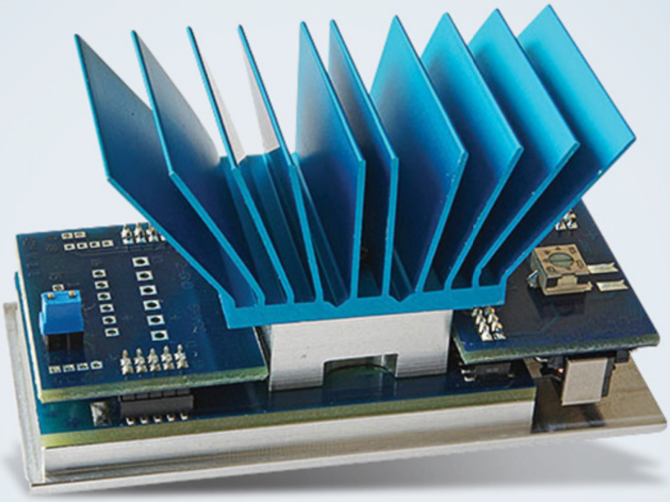
When security is required, Green Hills Software's INTEGRITY RTOS technology is the only option.

Green Hills®
SOFTWARE, INC.

www.ghs.com

The Newest Products For Your Newest Designs

Reap the benefits of energy harvesting power.

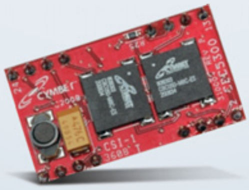


micropelt

TE-Power PLUS Thermal Energy Harvesting Evaluation Kit
mouser.com/micropelt/a

Texas Instruments
Authorized Distributor

Solar Energy Harvesting Development Tool: eZ430-RF2500-SEH
mouser.com/ti_ez430_rf2500_seh/



CYMBET™ CORPORATION

EnerChip™ EH CBC5300 Module
mouser.com/cymbet_cbc5300

AdaptivEnergy
Joule-Thief™ Energy Harvesting

Random Vibration Joule-Thief™ Module
mouser.com/adaptivenergy_joule-thief

WARNING: Designing with Hot, New Products May Cause A Time-to-Market Advantage.

Get your design ideas going with the latest renewable energy harvesting trends in electronics. Experience Mouser's time-to-market advantage with no minimums and same-day shipping of the newest products from more than 400 leading suppliers.

MOUSER® ELECTRONICS
a tti company

mouser.com **(800) 346-6873**

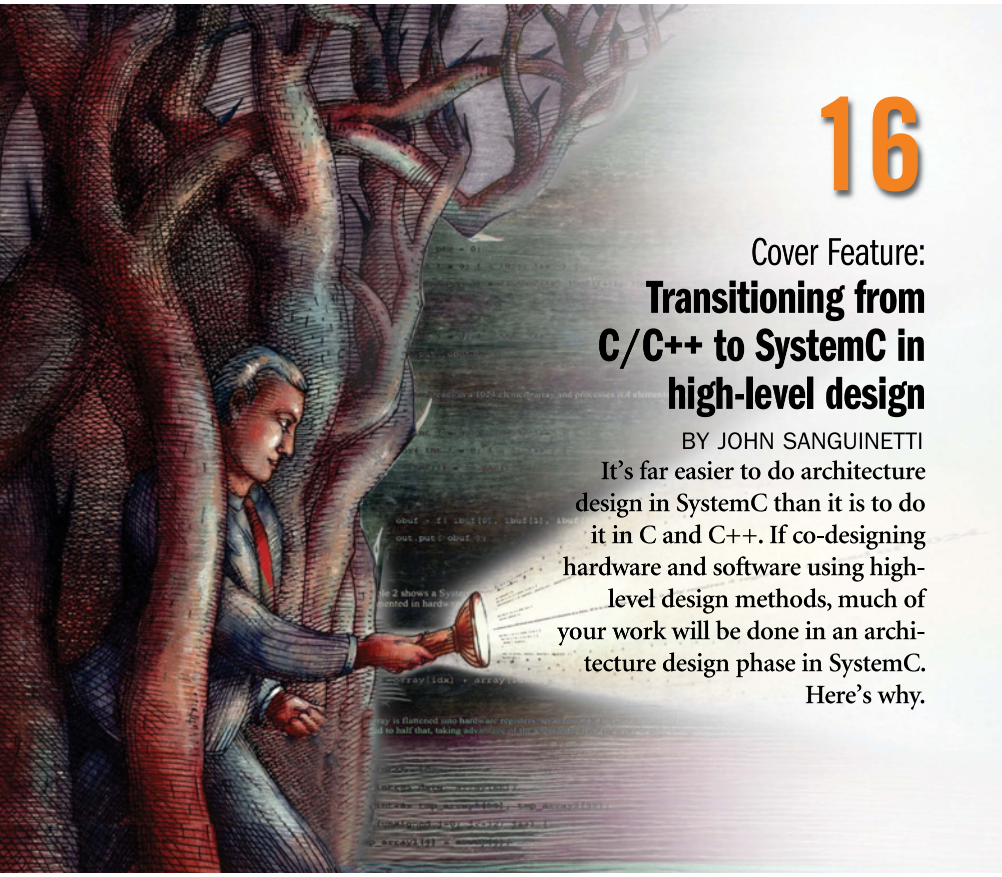
THE OFFICIAL PUBLICATION OF THE EMBEDDED SYSTEMS CONFERENCES AND EMBEDDED.COM

Learn today. Design tomorrow.

ESD

EMBEDDED SYSTEMS DESIGN

VOLUME 23, NUMBER 5
JUNE 2010



EMBEDDED SYSTEMS DESIGN (ISSN 1558-2493) print; (ISSN 1558-2507 PDF-electronic) is published 10 times a year as follows: Jan/Feb, March, April, May, June, July/August, Sept., Oct., Nov., Dec. by the EE Times Group, 600 Harrison Street, 5th floor, San Francisco, CA 94107, (415) 947-6000. Please direct advertising and editorial inquiries to this address. SUBSCRIPTION RATE for the United States is $55 for 10 issues. Canadian/Mexican orders must be accompanied by payment in U.S. funds with additional postage of $6 per year. All other foreign subscriptions must be prepaid in U.S. funds with additional postage of $15 per year for surface mail and $40 per year for airmail. POSTMASTER: Send all changes to EMBEDDED SYSTEMS DESIGN, P.O. Box 3404, Northbrook, IL 60065-9468. For customer service, telephone toll-free (877) 676-9745. Please allow four to six weeks for change of address to take effect. Periodicals postage paid at San Francisco, CA and additional mailing offices. EMBEDDED SYSTEMS DESIGN is a registered trademark owned by the parent company, EE Times Group.

COLUMNS

DEPARTMENTS

IN PERSON

ESC Chicago
June 8–9, 2010
www.embedded.com/esc/Chicago

ESC India
July 21–23, 2010
www.esc-india.com/

ESC Boston
September 20–23, 2010
www.embedded.com/esc/boston

Embedded Live
October 20–21, 2010
www.embedded.co.uk

ESC Silicon Valley
May 2–5, 2011
www.embedded.com/esc/sv

ONLINE

www.embedded.com

BUILD it [Reliably]

With Express Logic's award-winning BenchX® IDE or use tools from over 20 commercial offerings including those from ARM, Freescale, Green Hills, IAR, Microchip, MIPS, Renesas, and Wind River.

RUN it [Fast]

With Express Logic's small, fast, royalty-free and industry leading ThreadX® RTOS, NetX™ TCP/IP stack, FileX® FAT file system, and USBX™ USB stack.

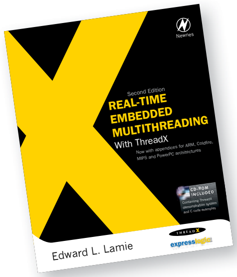
ANALYZE it [Easily]

With Express Logic's graphical TraceX® event analysis tool, and new StackX™ stack usage analysis tool. See exactly what is happening in your system, which is essential for both debugging and optimization.

SHIP it [Confidently]

No matter what "it" is you're developing, Express Logic's solutions will help you build it, analyze it, run it, and ship it better and in less time. Join the success of over 600,000,000 deployed products using Express Logic's ThreadX!

BENCHX THREADX TRACEX STACKX


Newnes

Second Edition

REAL-TIME EMBEDDED MULTITHREADING

With ThreadX

Now with appendices for ARM, Coldfire, MIPS and PowerPC architectures

CD-ROM INCLUDED

Containing ThreadX demonstration system and C code examples

THREADX

expresslogic

Edward L. Lamie

expresslogic

For a free evaluation copy, visit www.rtos.com • 1-888-THREADX

ThreadX, BenchX, TraceX and FileX are a registered trademarks of Express Logic, Inc. All other trademarks are the property of their respective owners.

ESD
EMBEDDED SYSTEMS DESIGN

Editorial Director
Richard Nass
(201) 288-1904
rich.nass@ubm.com

Managing Editor
Susan Rambo
susan.rambo@ubm.com

Contributing Editors
Michael Barr, John Canosa,
Jack W. Crenshaw, Jack G. Ganssle,
Dan Saks, Larry Mittag

Art Director
Debee Rommel
debee.rommel@ubm.com

European Correspondent
Colin Holland
colin.holland@ubm.com

Embedded.com Site Editor
Bernard Cole
bccole@acm.org

Production Director
Donna Ambrosino
dambrosino@ubm-us.com

Subscription Customer Service
P.O. Box 2165, Skokie, IL 60076
(800) 577-5356 (toll free)
Fax: (847) 763-9606
embeddedsystemsdesign@halldata.com
www.customerserviceesp.com

Article Reprints, E-prints, and Permissions
Mike O'Brien
Wright's Reprints
(877) 652-5295 (toll free)
(281) 419-5725 ext.117
Fax: (281) 419-5712
www.wrightsreprints.com/reprints/index.cfm?magid=2210

Publisher
David Blaza
(415) 947-6929
david.blaza@ubm.com

Editorial Review Board
Michael Barr, Jack W. Crenshaw,
Jack G. Ganssle, Bill Gatliff,
Nigel Jones, Niall Murphy, Dan Saks,
Miro Samek

EE|Times Group

Corporate—EE Times Group

Paul Miller	*Chief Executive Officer*
Felicia Hamerman	*Group Marketing Director*
Brent Pearson	*Chief Information Officer*
Jean-Marie Enjuto	*Financial Director*
Amandeep Sandhu	*Manager Audience Engagement*
Barbara Couchois	*Vice President Sales Ops*

UBM

Corporate—UBM LLC

Marie Myers	*Senior Vice President, Manufacturing*
Pat Nohilly	*Senior Vice President, Strategic Development and Business Administration*

BY Richard Nass

Android follows Linux's lead

I wrote a column about a year ago ("Android is coming where you least expect," *www.embedded.com/214501968*) talking about how the Android operating system (*www.android.com*) will have a place in applications other than handsets. When Google first developed the operating system (well, they purchased the company that developed the OS, but that's another story), it was clear that the company wanted to be a player in the handset space. And they've made some significant strides in that space (according to the lady sitting next to me on a recent flight, the Motorola Droid phone is the best thing ever invented).

It's not uncommon for one of the building blocks—Android in this case—to find its way into other areas of the embedded landscape. Embedded systems developers are a very resourceful group. If they think there's a better and more cost-effective way to produce a product, they're willing to give it a go.

I'm not endorsing Android here as the be-all, end-all OS for embedded devices, but it may be something that deserves a look. It's playing out in a similar fashion to where embedded Linux was a few years ago. Although Linux was touted as a so-so OS with a great price—free—we quickly learned that you get what you pay for. The investment in time and/or support proved to be higher than many people expected. Hence, Linux is used in lots of embedded applications, but it's far from omnipresent.

I expect Android to follow a similar path. We're now in the "let's see what it's all about" period. That fact was apparent at the recent Embedded Systems Conference in Silicon Valley. We held a half-day tutorial on how to get started in an embedded design using Android as your OS. The class was packed to say the least. The next day at ESC, I moderated a panel called "What it takes to build a device using Android." That, too, was packed. So there must be something to this phenomenon.

As was the case with Linux, some of the well-known operating system suppliers have either released or are readying a commercial version of Android or will support it in some fashion. That list includes Green Hills, Mentor Graphics, and Wind River. I expect, like Linux, it will have an impact, but only in certain niches. We'll have to wait and see how packed those classes are next year.

Rich Nass

Richard Nass
rich.nass@ubm.com

Richard Nass (rich.nass@ubm.com) is the editorial director of* Embedded Systems Design *magazine, Embedded.com, and the Embedded Systems Conference.

Royal Navy Astute Class nuclear-powered attack submarine.

Thales' periscope provides a 360° scan of the surface above with minimal risk of detection.

TDD and models for memory-mapped devices

This article (*Dan Saks, "Alternative models for memory-mapped devices," May 2010, p.9, www.embedded.com/224700534*) ties very nicely to the recent discussion about Test-Driven Development (see *www.embedded.com/224200702* and *www.embedded.com/224700535*). The relevance of the techniques that Dan recommends here is critical for TDD, but it is not obvious at first glance, so let me explain.

From both interviews with James Grenning about TDD, it is obvious that he recommends development and testing the code on the host as opposed to developing directly and exclusively on the embedded target. Also, he apparently means API-level fidelity of the code, so you compile the embedded code with a host compiler and run the code natively on the host. (This is contrast to much harder to achieve binary fidelity, where you compile the code with a cross-compiler for your embedded target and execute the binary in a simulator on the host.)

Now, to achieve API-level fidelity, you typically cannot access the device registers directly, because they don't exist on the host. Instead, you need to "plug-in" some other code in all these places. For example, instead of toggling an LED by writing to a register, on the host you might call a function to change a bitmap on your simulated display. And here is where the technique recommended by Dan comes in.

For example, the old-fashioned way of toggling an LED is shown in **Listing 1**. This code is difficult to replace with a function call on the host. In C++ you could use operator overloading, so the operation ^= could indeed become a function, but in C you can't.

In contrast, Dan recommends something like that shown in **Listing 2**. This code is very easy to re-implement on the host. So, the technique is not only safer, as Dan describes. It is also much more TDD-friendly.

! The technique is not only safer, as Dan describes. It is also much more TDD-friendly.

—Miro Samek
state-machine.com

See more comments on this article at *www.embedded.com/tigforums/thread.jspa?threadID=11191.*

Listing 1 The old-fashioned way of toggling an LED.

```
#define LEDS ((unsigned volatile *)0xFFFF6000)
. . .
LEDS ^= (1 << 3); /* toggle LED3 */
```

Listing 2 What Dan Saks recommends.

```
LEDS_registers * const LEDS = (LEDS_registers *)0xFFFF6000;
. . .
LEDS_toggle(3); /* toggle LED3 */
```

TDD for software and hardware

TDD makes a lot of sense, (*Jack Ganssle, An interview with James Grenning, Part 2, May 2010, p.33, www.embedded.com/224700535*) but why not use it for the software/hardware combination? Use of Boolean expressions to define the total system in a logical sense would make function trade-offs easy. Use of English for documentation/requirements is imprecise. By including the standard arithmetic operators, both control conditions and data manipulation are precisely defined. (Arithmetic operators are really Boolean macros.) In order to keep the microcycle short, a quick and easy Boolean simulator can run the tests.

The RTL simulators and use of HDL for hardware design do not fit well with TDD. It is ironic that tool flows are described using a flow diagram, but tools only use HDL input. About 30 years ago, IBM used programming flow charts to describe hardware and synthesis generated the hardware logic. Since then designers have been at the mercy of the synthesis program, and programmers have been at the mercy of the compiler/optimizer.

HDL was created before technology like GPS and Google maps so the old logic diagrams with the shaped block symbols were abandoned.

I am writing a simulator, but a GUI that shows a flow diagram like a street map rather than scrolling through source code would also help shorten the microcycle.

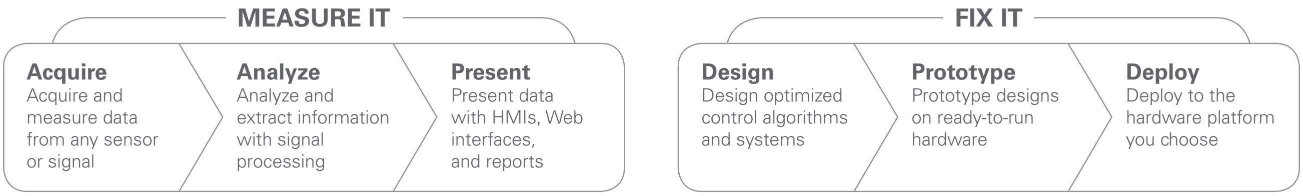
—KarlS
CEngine Developer

We welcome your feedback. Letters to the editor may be edited. Send your comments to Richard Nass at rich.nass@ubm.com or fill out one of our feedback forms online, under the article you wish to discuss.

336 Volts of Green Engineering

MEASURE IT – FIX IT

Developing a commercially viable fuel cell vehicle has been a significant challenge because of the considerable expense of designing and testing each new concept. With NI LabVIEW graphical programming and NI CompactRIO hardware, Ford quickly prototyped fuel cell control unit iterations, resulting in the world's first fuel cell plug-in hybrid.



MEASURE IT

Acquire
Acquire and measure data from any sensor or signal

Analyze
Analyze and extract information with signal processing

Present
Present data with HMIs, Web interfaces, and reports

FIX IT

Design
Design optimized control algorithms and systems

Prototype
Prototype designs on ready-to-run hardware

Deploy
Deploy to the hardware platform you choose

Ford is just one of many customers using the NI graphical system design platform to improve the world around them. Engineers and scientists in virtually every industry are creating new ways to measure and fix industrial machines and processes so they can do their jobs better and more efficiently. And, along the way, they are creating innovative solutions to address some of today's most pressing environmental issues.

>> Download the Ford technical case study at **ni.com/336**

800 258 7018

NATIONAL INSTRUMENTS™

By Jack W. Crenshaw

How I write software

In the two decades or so that I've been writing this column, we've covered a lot of topics, ranging from the best implementation of `abs(x)` to CRC algorithms to logic theory, Karnaugh maps, and electronic gates, to vector and matrix calculus to Fourier and z-transforms to control theory to quaternions, and lots more. I even wrote about whether or not the ancient Egyptians really built the values of π and ϕ into the Great Pyramid (they did. And $\sqrt{2}$, too).

Almost all of the topics involved math, but they also required software, either for embedded systems or just to illustrate a given algorithm. But I've never talked much about the way I write software. I thought the issue of my own personal software development methodology would be, at best, a distraction from the topic at hand and, at worst, a subject of great hilarity.

I've always known that the way I write software is different from virtually everyone else on the planet. Is this true for everyone? I don't have a clue.

My own personal software development methodology would be, at best, a distraction and, at worst, a subject of great hilarity.

That's why I thought it would be fun to share with you the way I write software. If you do it the same way, that's great. It would be nice to know that there are other folks who share my approaches. On the other hand, if it's not the same way you do it, we have three possible outcomes. Either you'll learn from me, and decide that some of my approaches have merit; I'll learn from you, and decide that I've been doing it wrong all these years; or my revelations will serve as a subject of great hilarity. Any of those outcomes, I submit, would be good. It's a win-win-win situation.

THE ENVIRONMENT

Let's begin with the software development environment. In my time, I've used some really rotten development environments. I've worked with systems whose only I/O device was a Teletype ASR-33. We got so tired of emptying the chad-catcher on the paper tape punch, we just let it flutter to the floor. (At the end of the day, guess who got to sweep it up.)

I worked on one program where my only "compiler" was a line-by-line assembler (meaning that it didn't accept mnemonic names, only absolute addresses). The bulk storage device was an audio cassette. Just so we're clear, this wasn't a homebrew system I cobbled up out of old Intel 4004s. paper clips, and chewing gum. It was a system built up by serious, if boneheaded, managers in a very large corporation.

Perhaps because of these nightmare environments, I really appreciate a good one. I especially love using integrated development environments (IDEs), in which all the tools are interconnected, preferably by a lovely and fast GUI interface. I know, I know, real men don't use GUIs or IDEs. Some folks much prefer to use only command-line interfaces. and to write ever more complex and inscrutable makefiles.

If that's you, more power to you. Whatever works. But I can't hide my view that such an approach is an affectation, involved more with earning one's credentials as a software guru than on producing good code. I'm sure that there are good, valid reasons to write a makefile rather than let the IDE do it for you. I just can't think of one, at the moment.

So what do I do when I'm told that I must use the environment the project managers already picked out? First of all, I'd ask them (always discretely, of course, in my usual, ultra-tactful manner) why on earth they chose the development environment without discussing it with those of us who had to use it. Second, I'd campaign mightily for an environment more suited to my approaches.

As an aside, I used to teach a short course in the development process for embedded microprocessors. I told the management folks that they should plan to give their soft-

Jack Crenshaw is a systems engineer and the author of Math Toolkit for Real-Time Programming. He holds a PhD in physics from Auburn University. E-mail him at jcrens@earthlink.net.

RENESAS

ware folks every possible computer tool available—hang the expense. I pointed out that software development is very labor-intensive, so the front-end cost of good tools would be quickly made up by increased programmer productivity.

Needless to say, sometimes I'm successful in changing the managers' minds. More often, I'm not. So what do I do then? I make do as best I can, but I discreetly tweak the system and my approach to make the environment behave more like an IDE. For example, I might add script files, hot keys, etc., so that pressing a single button would invoke one tool from another.

REQUIREMENTS

Crenshaw's Law #42 says: Before setting out to solve a problem, find out what the problem is. Law #43 says design it before you build it, not after.

In the software world, this means performing requirements analysis. Write down, in as much detail as you can, what problem you want the software to solve, how you want it to solve the problem, and how you want people to interface with it.

In the 1980s, the popular project plan was to start with an analysis of the system requirements. Then you decompose into software requirements, design requirements, etc. At the end of each phase, you hold a formal review before moving onto the next step. No fair skipping ahead.

They called this process the "waterfall" approach. If you were writing a formal proposal to NASA, DoD, or whoever, a clever picture of the waterfall could win you the contract. In the waterfall diagram, the "code and unit test" phase of the plan was usually the next-to-last step in the process (the last being integration and test). And woe be upon anyone who skipped ahead to writing code.

Today, the waterfall approach has been completely discredited and deemed ineffective. Personally, I think the only reason it seemed to work was that nobody actually followed it. Some of us practitioners broke the rules, and skipped ahead to try out our ideas on the side. The whole thing was a sham. We had slick-looking and impeccably dressed salespersons giving formal presentations to the customer, showing them how wonderfully the waterfall approach was working, and assuring the customer that we were 90% complete. Meanwhile, a few of us slaved away in a secret back room, using approaches that actually worked.

But those were the bad old days. Today, the hot keywords are spiral development, prototyping, incremental and agile approaches, and extreme programming (XP). I'm a big fan of XP, though I'm not sure that what I do fits the mold.

Today, no one in their right mind would still support the waterfall approach, right? Well, I don't know. In 1984 our software managers had a rule: you were not allowed, on penalty of—something bad—to even *run* the compiler, until your software had passed code review. Even then, the compiler was never to be used for developing test drivers and supporting unit testing. It was only to be used on the entire system build, after all integration had been done.

You should go back and reread that paragraph, to really get your arms around the concept. We had a nice, and relatively modern, timeshare system (Unix), with a nice compiler, programming editor, debugger, and other aids, but we weren't allowed to use the compiler. We were perfectly free to use the editor to write the code but only to document our designs for the code reviews. We were expressly forbidden to actually run the compiler. How brilliant was *that*?

But that was 25 years ago. We don't do anything so ridiculous now, right? Well, let me see . . . .

On a more recent project, I heard a manager berate a colleague for actually writing code before the requirements review. If he had been working in his old company, the manager said, he would have *fired* the guy for such a violation of the rules.

That was way back in . . . let me see . . . 2008.

I won't say much more about project planning or about requirements analysis here. In part, it's because using my approach, the requirements evolve with the code. Call it spiral development, if you like.

I START WITH CODE

The method I use to develop software may surprise you. In the worst possible tradition of the anti-waterfall plan, I jump right into code. I sit down at the keyboard, flex my fingers, crack my knuckles, and type:

```
void main(void){}
```

That's right: I write the null program. I do this on every new start. If I'm in a really adventurous mood, I'll add:

```
cout << "Hello, Jack!" << endl;
```

and maybe even:

```
x = 2; y = 3;
cout << x + y << endl;
```

This is not a joke. I really, really do this. Every time.

Why? I think it's just my way of assuring myself that the genie behind the glass screen is still awake, and still doing his job. And my computer and its operating system haven't gone berserk overnight (not all that unlikely, these days),

Mostly, I do it to get myself in the mode of expecting success, not failure.

Several years ago, educators came up with the notion of "programmed learning." The idea was to write the textbook very much like a computer program, complete with loops and goto's. After teaching a few facts (never more than three or so), the text asks a series of questions. If you give the right answers, you get to skip to the next section. Otherwise, you may be directed to loop back to the beginning of the section, and read it again. Programmatically, this is the familiar:

```
while(1){...}
```

infinite loop structure. If you're particularly dense, you could be stuck here forever.

Alternatively (and better), you might be directed to a separate section, which explains the facts in more detail. It's a sort of hypertext. Ideally, the writers of the text were smart enough to tell when a given student needed extra instruction.

The inventors of programmed learning were careful to point out that those writing such textbooks should give the information in very small steps. You don't write a whole chapter on, say, the quadratic formula, and save all the questions for the end of the chapter. You teach a very few simple concepts—two or three at most—and ask the questions immediately.

The reason, they say, is that people need lots of positive feedback. If they get too many answers wrong, they get discouraged and give up. By giving them easy questions often, you give them lots of warm fuzzies, and make the study seem a lot more fun than drudgery. Everybody knows that people like to succeed more than to fail, and they like the feeling that they can actually complete the task. The notion of programmed learning capitalizes on these natural tendencies.

I think that's why I start with the null program. I like my warm fuzzies early and often.

When I'm writing code, I want to expect success. I'm mildly surprised if a bit of code doesn't compile without error, the first time. I'm shocked to my core if it compiles, but doesn't run properly, or doesn't give the right answer.

Some programmers, I submit, have never experienced that feeling. Certainly not those poor souls that had to write and integrate the entire system before even trying to compile it. How certain of success would *they* be?

THE BIOLOGICAL ANALOGY

Think of a human egg cell. Once fertilized, it starts dividing, first into two cells, then four, then sixteen, and so on. The egg has become a zygote. Early on, every part looks like every other one. Later, they will differentiate, some cells becoming skin cells, some nerve cells, some heart and muscle cells. But in the early stages, they're completely undifferentiated. The organism may ultimately become a lizard, a fish, a bird, or a fern, but at this point, all the cells are alike.

I think of my null program as the software equivalent of a zygote. At the early stages, they can be any program at all. That's why I don't sweat the requirements too much. The null program and its cousins will satisfy any set of requirements, to some degree of fidelity. The requirements will flesh out—differentiate—along with the program that satisfies them.

As I continue the software development process, I stick to the philosophy of programmed learning. I never write more than a few lines of code, before testing again. I'm never more than mildly surprised if they don't work. I guess you could call this approach "programmed programming."

THEORY VS. PRACTICE

I have another reason to start coding early: I don't know exactly how the program should be structured. Like the potential fish or duck, many things are not clear yet. In the old, discredited waterfall approach, this was not a problem. We were supposed to assume that the guys who wrote the top-level requirements document got it perfect the first time. From then on, it was simply a matter of meeting those requirements.

To be fair, some visionaries of this approach had the sense to add feedback loops from each phase, back to the previous one. This was so that, in the completely improbable off chance that someone made a mistake further back upstream, there was a mechanism to fix it. Practically speaking, it was almost impossible to do that. Once a given phase had been completed, each change required a formal engineering change request (ECR) and engineering change notice (ECN). We avoided this like the plague. The guys in the fancy suits just kept telling the customer that the final product would meet the original spec. Only those of us in the back room knew the truth.

This is why I like the spiral development approach, also known as incremental, or prototype approaches. As the program evolves, you can still tell the customer, with a straight face, not to worry about all these early versions of the program. They are, after all, only prototypes. He doesn't need to know that one of those prototypes is also the final product.

TOP DOWN OR BOTTOM UP?

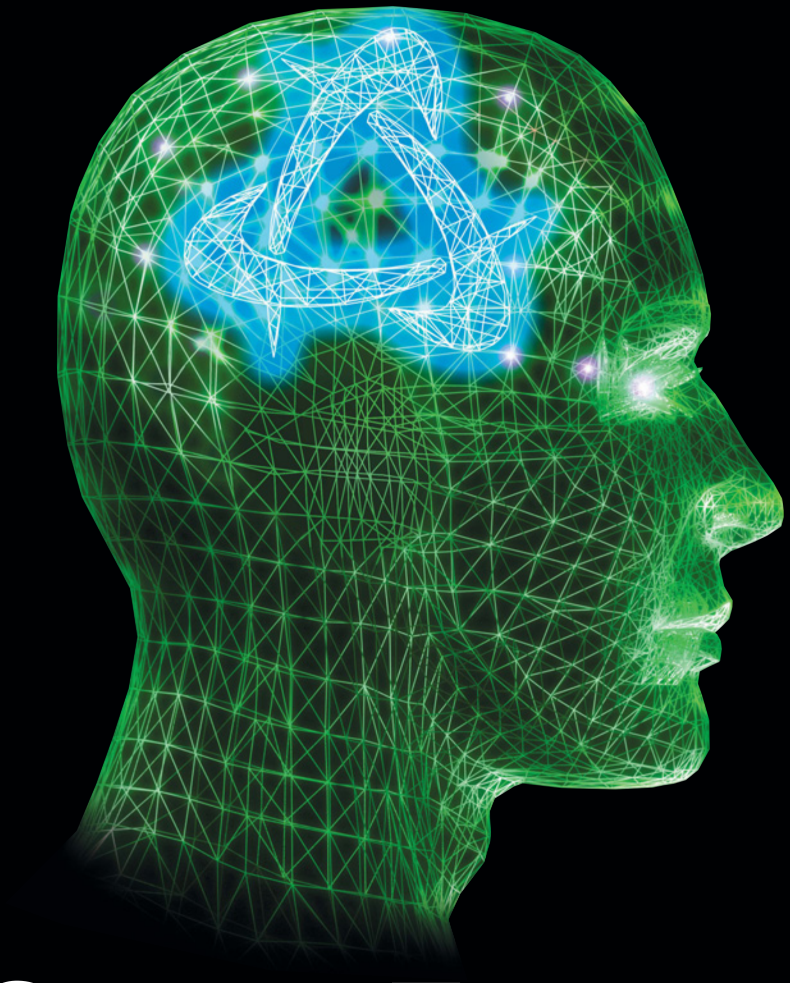
The great racing driver, Stirling Moss, was once asked if he preferred a car with oversteer or understeer. He said, "It really doesn't matter all that much. In the end, it just depends on whether you prefer to go out through the fence headfirst, or tailfirst." Ironically, Moss had a terrible crash that ended his career. He went out through a literal, wooden fence, headfirst.

Software gurus have similar differences of opinion on bottom-up vs. top-down development. Purists will claim that the only way to build a system is top down. By that, they mean design and build the outer layer (often the user interface) first. Put do-nothing stubs in place until they can be fleshed out. Even better, put in "not quite nothing" stubs that return the values you'd expect the final functions to return.

Other practitioners prefer a bottom-up approach, where you build the lowest-level code—sometimes the hardware interfaces—first. Connect them all together properly, and you've got a working program.

Paraphrasing Moss's comment, it all depends on how you prefer your project to fail. You can start with a beautifully perfect top-down design, only to discover in the end that your software is too slow or too big to be useful. Or you can start bottom-up, with a lot of neat little functions, only to discover that you can't figure out how to connect them together. In either case, you don't know until the very end that the system is not going to work.

That's why I prefer what I call the "outside in" approach. I start with both a stubbed-out main program *and* the low-level



functions I know I'm going to need anyway. Quite often, these low-level functions are the ones that talk to the hardware. In a recent column, I talked about how we wrote little test programs to make sure we could interface with the I/O devices. It took only a day, and we walked away with the interface modules in hand. During the course of the rest of the project, it's a good feeling to know that you won't have some horrible surprise, near the end, talking to the hardware.

After all is said and done, there's a good and game-changing reason (I assert) that a pure top-down process won't work. It's because we're not that smart. I've worked on a few systems where I knew exactly what the program was supposed to do before I began. Usually, it was when the program was just like the previous four. But more than once, we've not been able to anticipate how the program might be used. We only realized what it was capable of after we'd been using it awhile. Then we could see how to extend it to solve even more complex problems.

In a top-down approach, there's no room for the "Aha!" moment. That moment when you think, "Say, here's an idea." That's why I much prefer the spiral, iterative approach. You should never be afraid to toss one version and build a different one. After all, by then you've got the hang of it, and much of the lower-level modules will still be useful.

The top-down approach isn't saved by object-oriented design (OOD), either. One of the big advantages of OOD is supposed to be software reusability. In the OOD world, that means *total* reusability, meaning that you can drop an object from an existing program, right into a new one, with no changes. I submit that if you follow a top-down approach to OOD, you'll never be able to achieve such reusability. To take a ridiculous example, I might need, in a certain program, a vector class that lets me add and subtract them easily. But for this program, I don't need to compute a cross product. So unless I look ahead and anticipate future uses for the class, it'll never have the cross product function.

KISS

I've been using modular designs and information hiding since I first learned how to program. It's not because I was so much smarter than my coworkers; it's just the opposite: I'm humble (and realistic) enough to know that I need to keep it short and simple (KISS). It always amazes me how little time it takes for me to forget what I wrote a few weeks ago, let alone a few years. The only way I can keep control over the complexity inherent in any software effort, is to keep the pieces small enough and simple enough so I can remember how they work, simply by reading the code.

I was also lucky in that the fellow who taught me to program never bothered to show me how to write anything *but* modules. He had me writing Fortran functions that took passed parameters, and returned a single (possibly array) result. By the time I found out that other people did things differently, it was too late. I was hooked on small modules.

The term "modularity" means different things to different people. At one conference, a Navy Admiral gave a paper on the Navy's approach to software development. During the question and answer period, I heard this exchange:

Q: In the Navy programs, do you break your software up into modules?
A: Yes, absolutely, In fact, the U.S. Navy has been in the forefront of modular design and programming.
Q: How large is a typical module"
A: 100,000 lines of code.

When I say "small," I mean something smaller than that. A *lot* smaller. Like, perhaps, one line of executable code.

In my old Fortran programs, I was forever having to convert angles from degrees to radians, and back. It took me 40 years to figure out: Dang! I can let the computer do that. I wrote:

```
double radians(double x){
  return pi*x/180;
}

double degrees(double x){
  return 180*x/pi;
}
```

Gee, I wish I'd thought of that sooner! (For the record, some folks prefer different names, that make the function even more explicit. Like `RadiansToDegrees`, or `Rad2Deg`, or even `RTD`). One colleague likes to write `ConvertAngleFromRadiansToDegrees`. But I think that's crazy. He's a weird person.

Back in my Z80 assembly-language days, I had a whole series of lexical-scan functions, like `isalpha`, `isnum`, `isalnum`, `iscomma`, etc. The last function was:

```
iscomma:    cpi  ','
            ret
```

A whole, useful function in three bytes. When I say "small," I mean *small*.

Is this wise? Well, it's sure worked for me. I mentioned in a recent column (It's Too Inefficient) that I've been berated, in the past, for nesting my functions too deeply. One colleague pointed out that "Every time you call a subroutine, you're wasting 180 microseconds."

Well, at 180 microseconds per call (on an IBM 360), he might have had a point. The 360 had no structure called a stack, so the compiler had to generate code to implement a function call in software.

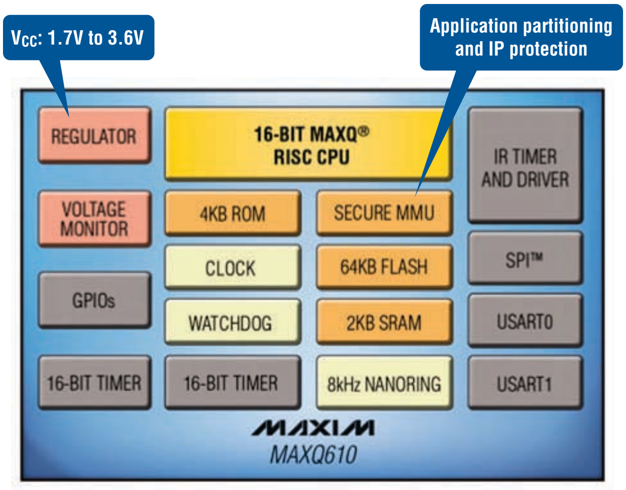
But certainly, microprocessors don't have that problem. We do have a stack. And we have built-in `call` and `return` instructions. The `call` instruction says, simply, push the address of the next instruction, and jump. `return` means, pop and jump. On a modern CPU, both instructions are ridiculously

CONTINUED ON PAGE 33

Extend battery life with the MAXQ610 16-bit microcontroller

Up to 12 MIPS in under 4mA—industry's highest MIPS/mA!

The MAXQ610 microcontroller is designed for low-cost, high-performance, battery-powered applications. This 16-bit, RISC-based microcontroller has a wide operating range (down to 1.7V) for long battery life and ultra-low power consumption. Its anticloning features and secure MMU enable you to protect your IP.



Microcontroller

- 16-bit MAXQ RISC core
- 64KB flash memory, 2KB SRAM
- Ultra-low supply current
  - Active mode: 3.75mA at 12MHz
  - Stop mode: 200nA (typ), 2.0µA (max)
- Wide, 1.7V to 3.6V operating voltage range
- IP protection
  - Secure MMU supports multiple privilege levels, helps protect code from copying and reverse engineering

Peripherals

- Two USARTs and one SPI master/slave communication port
- Two 16-bit timers/counters
- 8kHz nanoring functions as programmable wakeup timer
- IR timer simplifies support for low-speed infrared communication
- IR driver capable of 25mA sink current

Part	Temp Range (°C)	Program Memory	Data Memory	Operating Voltage (V)	Package	Price† ($)
MAXQ610B	0 to 70	64KB flash	2KB SRAM	1.7 to 3.6	40-TQFN	1.45

MAXQ is a registered trademark of Maxim Integrated Products, Inc.
SPI is a trademark of Motorola, Inc.
†1000-up recommended resale. Prices provided are for design guidance and are FOB USA. International prices will differ due to local duties, taxes, and exchange rates. Not all packages are offered in 1k increments, and some may require minimum order quantities.

www.maxim-ic.com/MAXQ610-info

MAXIM DIRECT
www.maxim-ic.com/shop

AVNET electronics marketing
www.em.avnet.com/maxim

MAXIM
INNOVATION DELIVERED™

For free samples or technical support, visit our website.

_ptr = 0;
r i = 0; i
buf ptr++]
int i = 0; i
.put(obuf[i]
ment reads in a 102
int j = 0; j
or(int i = 0;
ibuf[i] = in.ge
}
obuf = f(ibuf[0]
out.put(obuf);
}
ple 2 shows a System
mented in hardwa
array[idx] + arr
rray is flattened into hardwa
ed to half that, taking advan
uint<5> idx;
uint<8> data, array
uint<8> tmp_array1[
(unsigned j=0; j<=
p_array1[i]

It's far easier to do architecture design in SystemC than it is to do it in C and C++. If co-designing hardware and software using high-level design methods, much of your work will be done in an architecture design phase in SystemC. Here's why.

Transitioning from C/C++ to SystemC in high-level design

BY JOHN SANGUINETTI

In high-level design, high-level code is put through a series of steps on its way to becoming register transfer level (RTL) code. The first step, algorithm design, is usually done in C or C++, where the high-level code that describes how the system will function is created. To be implemented in hardware, this high-level code must be converted to RTL code, using a synthesis tool. It's almost never the case, however, that high-level synthesis using the result of the algorithm design phase will produce a desirable RTL implementation. An architecture design phase that precedes high-level synthesis is required in order to produce RTL code with the desired characteristics.

Making a translation to SystemC for this step has become the preferred high-level design method. In this article, I'll give some examples of steps taken in the architecture design phase that can help you achieve good RTL code.

High-level design has many advantages over the more commonplace design flow that begins with RTL code. Among the most compelling advantages is the improved verification efficiency that a higher level of abstraction offers. It's apparent to the

High-level design flow.

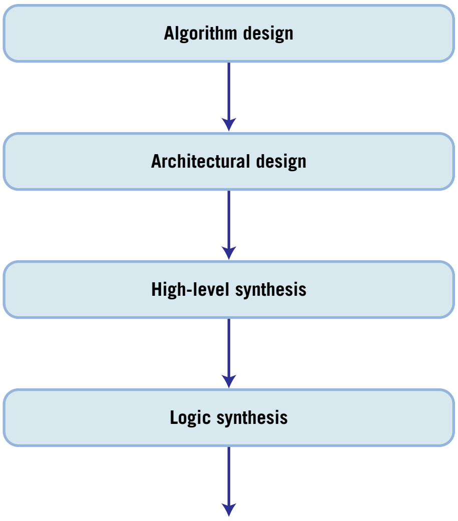


Figure 1

point of being self-evident that when the source code of a design is created, fewer errors occur if the source is at a higher abstraction level than if it is at a lower level. However, a process is still required to verify the transformations that are applied to the design description as it proceeds through the design flow from creation to final realization.

Figure 1 shows the initial steps in the high-level design (HLD) flow. **Figure 2** shows the flow with the verifica-

Detailed high-level design flow.

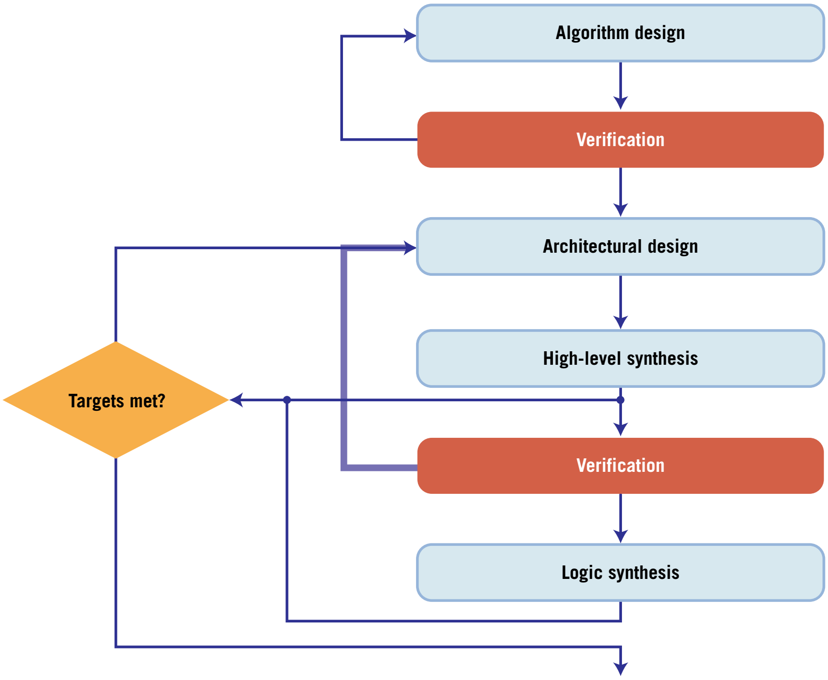


Figure 2

tion steps and the design loops added. A verification step after each design step can result in a loop back to fix a design error. Another loop after each of the two synthesis steps can return you to the architecture design step, where you'll improve whatever relevant design criteria have not been met. This loop back to the architecture design step is a key part of the high-level design process.

ARCHITECTURE DESIGN

The architecture design step in **Figure 1** is the fundamental "hardware design" step in a high-level design flow. High-level design, particularly when done using "Plain old C," is often presented as a process where the designer simply takes a C algorithm, runs it through a high-level synthesis tool, and gets high-quality RTL. This high-quality result, however, almost never happens.

The architecture design step is where hardware design is done for a fundamental reason. Algorithms developed in C are software representations, and there are very different costs and benefits between hardware and software implementations.

Some of the fundamental differences between software and hardware implementations are:

- Hardware is hierarchical, and the communication between subunits (subroutines in software, modules in hardware) is different. While only a few calling conventions implement software interfaces, there are many more possibilities for hardware interfaces. Plain old C (PoC) provides call by value, which, along with the ability to pass pointers, suffices for subroutine calls. However, there is no facility for representing the myriad variations of hardware interfaces. For example, ready-valid, trigger-done, AMBA bus read/write, custom bus—there are often good reasons for a hardware implementation to use a custom interface.
- The cost of memory access in hardware is much higher than in software. Consequently, PoC algorithms often simply store a lot of data in memory, then access it in whatever fashion is convenient for the algorithm. By contrast, the same algorithm implemented in hardware can often be done with a much smaller window of the data being processed at a time, and either written once or passed on to another processing phase. Image processing algorithms are usually done this

Example 1

```
for( int i = 0; i < 1024; i++ )
  ibuf[i] = in.get();

int obuf_ptr = 0;
for( int i = 0; i < 1024; i+=4 ) {
  obuf[obuf_ptr++] = f( ibuf[i], ibuf[i+1], ibuf[i+2], ibuf[i+3] );
}

for( int i = 0; i < 256; i++ )
  out.put( obuf[i] );
```

This code fragment reads in a 1024 element array and processes it four elements at a time. If it is rewritten as below, it only requires four registers, instead of 1024.

```
for( int j = 0; j < 1024; i+=4 ) {
  for( int i = 0; i < 4; i++ ) {
    ibuf[i] = in.get();
  }

  obuf = f( ibuf[0], ibuf[1], ibuf[2], ibuf[3] );
  out.put( obuf );
}
```

Example 2

```
sc_uint<5> idx;
sc_uint<8> data, array[64];
data = array[idx] + array[idx+32];
```

The array is flattened into hardware registers, so accessing it with a variable index requires two 64-to-1 8-bit muxes between the registers and the adder. By rewriting the code as follows, the muxing can be reduced to half that, taking advantage of the knowledge that the array is only being accessed in two contiguous subsets. The loop will simply turn into a set of wires.

```
sc_uint<5> idx;
sc_uint<8> data, array[64];
sc_uint<8> tmp_array1[32], tmp_array2[32];
for (unsigned j=0; j<=32; j++) {
  tmp_array1[j] = array[j];
  tmp_array2[j] = array[j+32];
}
data = tmp_array1[idx] + tmp_array2[idx+32];
```

way. The algorithm in software will simply iterate over the whole image, while the hardware implementation will work on a smaller window that moves over the image, as **Example 1** demonstrates.)

- Conditional expressions are cheap in software but can be expensive in hardware. This is illustrated when accessing array elements with a variable index when the array is mapped onto dedicated hardware registers, shown in **Example 2**.

! The same algorithm implemented in hardware can often be done with a much smaller window of the data being processed at a time.

DESIGN OPTIMIZATION

It's convenient to use a different language for the architecture design step rather than Plain old C, because C does not provide the ability to represent a few key hardware-related features, notably hierarchy, data-path widths, and concurrency. SystemC was developed for just this reason. SystemC is not a separate language, but simply a C++ class library that provides the missing features.

C can't represent a few key hardware-related features, notably hierarchy, data-path widths, and concurrency. SystemC was developed for just this reason.

It's far easier to do architecture design in SystemC than it is to do it in C.

Example 1 shows a C algorithm as it would be written for software implementation and how it would be modified for hardware implementation. Making the change to SystemC from the original C is a small step compared with the iterative process that takes place while refining the hardware implementation.

Example 3

```
void matmult::matrix_multiply() {
    for (sc_uint<3> i=0; i<DIM; i++)
      for (sc_uint<3> j=0; j<DIM; j++)
                                      elementloop(i,j);
}

#if FASTER
void matmult::elementloop(sc_uint<3> i, sc_uint<3> j) {
    Mtype temp_c[DIM];
    for (sc_uint<3> k=0; k<DIM; k++) {
      temp_c[k] = A[i][k] * B[k][j];
    }
    C[i][j] = temp_c[0]+ temp_c[1]+ temp_c[2]+ temp_c[3];
}
#else // smaller
void matmult::elementloop(sc_uint<3> i, sc_uint<3> j) {
    C[i][j] = 0.0;
    for (sc_uint<3> k=0; k<DIM; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
}
#endif

void matmult::thread0() {
    do_reset();
    while (true) {
      read_data();                    // reads the A and B matrices
      matrix_multiply();
      write_data();                   // writes the resulting C matrix
    }
}
```

Example 2 shows a SystemC fragment that adds two elements from an array, where each operand is half the array apart. The straightforward original code results in two big muxes when implemented in hardware. The less obvious modified code results in a much smaller hardware implementation.

The examples show transformations made to source code to produce a better hardware result out of high-level synthesis. These are examples of optimization where the modified code will result in a smaller, more efficient hardware implementation when high-level synthesis is applied. In many cases, the design process is one of successive optimizations in order to get a hardware implementation that is as small, fast, and/or power efficient as possible. This is a manual process, where the hardware designer uses his knowledge and experience to make the appropriate code transformations.

This is a manual process, where the hardware designer uses his knowledge and experience to make the appropriate code transformations.

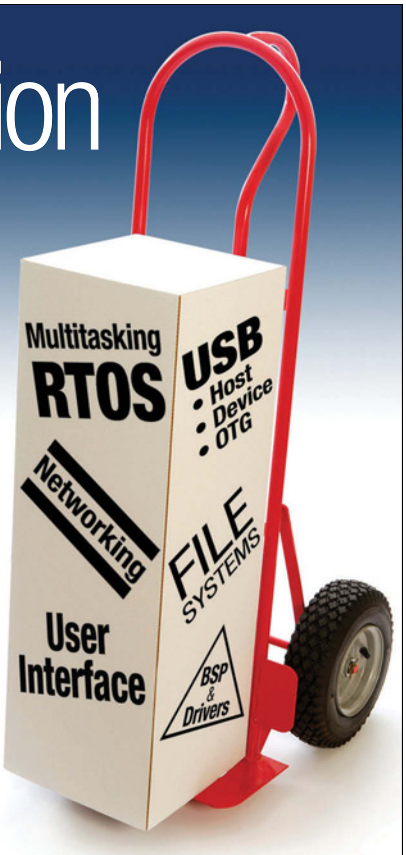
One can ask why such a manual process is necessary, since the promise of high-level synthesis is to remove the drudgery of mapping higher-level operations to hardware. The answer is that there are many ways to implement an algorithm in hardware, and no single choice is the best under all design constraints. Some transformations, it could be argued, should always be done, and could thus be implemented in the high-level synthesis tool, as in Example 2. However, there are a great many examples of code where the high-level synthesis tool could in principle recognize the original without requiring the code to be rewritten but in practice requires the designer to do the transformation. No high-level synthesis tool will recognize them all (at least within our lifetimes), which means the hardware designer will always have a job.

DESIGN EXPLORATION

The architecture design step is also the step where design exploration is done. Exploration has been recognized as one of the great benefits of doing high-level design, since it's relatively easy to control the high-level synthesis tool to produce implementations that have different properties, typically area, latency, or power consumption. We usually think of this process as adding either global or local directives to the synthesis tool. Examples of global directives are "unroll all the loops" and "flatten all the arrays." An example of a local directive would be "set the latency of this block to be three cycles." It's easy to see that design exploration can be accomplished by changing these directives, for example, "unroll loop A but not loop B," "map arrays to memories," or "set the latency to five cycles."

However, there are occasions where changing the source code can be useful for design exploration. **Example 3**, a 4×4 matrix multiply, shows such a case.

This is a 4×4 matrix multiplication, where the data type of the matrix elements is "Mtype." Mtype could be changed from integer to fixed point to floating point, and this code would not change. The conditional code ("`#if FASTER ... #else ... #endif`") shows two different implementations of the inner-most loop. In the first version of `elementloop`, a multicycle pipelined part can be created that will



do four multiply operations and three additions, producing a resulting matrix element every cycle, after an initial latency. This code required hard-coding the three addition operations, so if `DIM` were a value other than four, that line of code would have to be changed. The second version of `elementloop` results in a much smaller implementation, using a single multiply and addition that will produce a resulting matrix element every four cycles. The synthesis directives to unroll the loops, flatten the arrays, pipeline the inner loops, and create custom datapath functional units are not shown.

Example 3 shows two alternatives to writing the inner loop of the matrix multiply, and it shows the difference between writing code as software and writing it for hardware implementation. Executed on a standard processor, both of these versions would have the same performance, since there is only one multiplier and one adder available. However, when using high-level synthesis, the synthesis tool is able to use more than one multiplier and more than one adder in the first case. In the second case, the synthesis tool would use just one multiplier and one adder (or a combined multiply-accumulate functional unit).

A NEW STEP TOWARD QUALITY

The architecture design step is where hardware design is done. The designer uses this step to experiment with different RTL architectures, and then uses it to optimize the implementation after the architecture has been decided. Both architecture exploration and optimization can be done by varying the synthesis directives or by changing the high-level source code. ■

! No high-level synthesis tool will recognize them all (at least within our lifetimes), which means the hardware designer will always have a job.

John Sanguinetti is chief technology officer at Forte Design Systems and has been active in computer architecture, performance analysis, and design verification for 20 years. After working for DEC, Amdahl, ELXSI, Ardent, and NeXT computer manufacturers, he founded Chronologic Simulation in 1991 and was president until 1995. He was the principal architect of VCS, the Verilog Compiled Simulator and was a major contributor to the resurgence in the use of Verilog in the design community. Dr. Sanguinetti served on the Open Verilog International Board of Directors from 1992 to 1995 and was a major contributor to the working group which drafted the specification for the IEEE 1364 Verilog standard. He was a cofounder of Forte Design Systems. He has 15 publications and one patent. He has a Ph.D. in computer and communication sciences from the University of Michigan.

ADDITIONAL READING:

- Black, D. and J. Donovan. *SystemC: From the Ground Up*. Springer, New York, NY. 2006.
- Kurup, P., T. Abbasi, and R. Bedi. *It's the Methodology, Stupid!* ByteK Designs, Inc. Palo Alto, CA. 1998.

Minimax fitting is often preferred to regression when the goal is to find a fitting line with minimum tolerance, such as some automatic control applications. Here's part 1 of 3 on line-fitting algorithms.

Line-fitting algorithms for exceptional cases—minimax line fitting

BY CHRIS THRON

In many embedded systems design applications, line-fitting techniques, particularly minimax algorithms, are used instead of the more well-known regression methods to "fit" noisy data. More often than not, minimax fitting is preferred to regression when the goal is to find a fitting line with minimum tolerance, such as some automatic control applications.

My first exposure to minimax line fitting came via an engineer who asked me to help him fit some sampled sensor data. He was getting unacceptable results with regression and thought that minimax might do better. The engineer told me that he wanted something quick and simple and that small code size was required, but that high performance was not a priority because the computations would be done off-line.

This article presents my own attempts to meet these requirements. It describes both useful strategies and pitfalls to avoid in tackling similar problems.

REGRESSION DOES NOT ALWAYS GIVE THE "BEST" LINEAR FIT

By far the most common way to fit noisy data with a line is to use simple regression. If the noisy data consists of points (x_n,y_n), $n=1,\ldots,N$, then the regression equations give:

$Y = M\bullet X + B$

where M and B are defined in terms of the accessory values Σ_X, Σ_Y, Σ_{XX}, and Σ_{XY} as follows:

Σ_X = the sum of the x-values = $\Sigma_n x_n$
Σ_Y = the sum of the y-values = $\Sigma_n y_n$
Σ_{XX} = the sum of the x^2-values = $\Sigma_n x_n^{\ 2}$
Σ_{XY} = the sum of the xy-values = $\Sigma_n x_n y_n$
$M = (N\bullet\Sigma_{XY} - \Sigma_Y\ \Sigma_X) / (N\bullet\Sigma_{XX} - \Sigma_X\ \Sigma_X)$
$B = (\Sigma_Y - M\bullet\Sigma_X) / N$.

There are many cases where regression is not the best option: when a minimax fit is required or when noise varies and affects the linear trend

Nonetheless, there are many cases where regression is *not* the best option, for instance when:

- a *minimax fit* is required (in other words, one that minimizes the maximum deviation between line and data);
- the noise varies systematically in such a way that it affects the linear trend;
- the linear fitting is taking place real-time and is continually adjusted as new data comes in;
- there is a possible alias in the y-value (for instance with angle data, there may be an ambiguity of $\pm k\bullet\pi$ or $\pm 2k\bullet\pi$);

Comparison of regression and minimax line fit to noisy data.

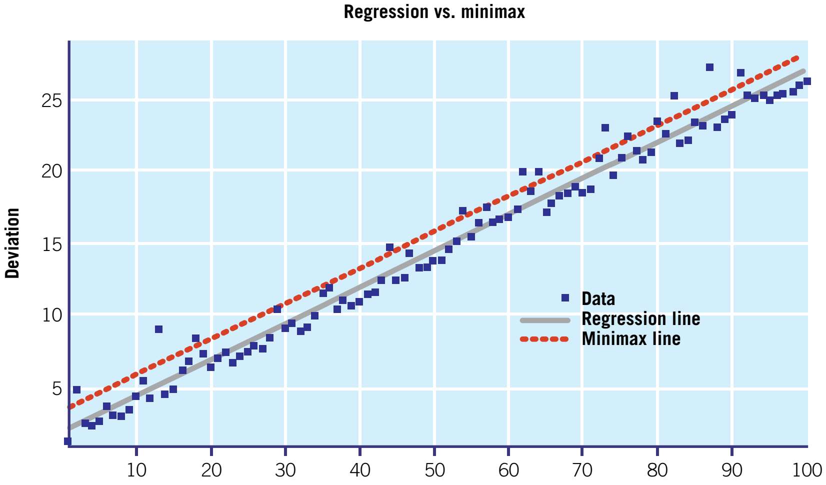


Figure 1

Web libraries where you can download the Remez algorithm.

Web library address	Free	Languages	Program documentation	Algorithm documentation
www.netlib.org	Yes	Algol	Gives literature references	Gives literature references
www.boost.org	Yes	C++	Yes	Yes (contains inaccuracy)
www.mathworks.com	Yes	Matlab	Yes	Yes (excellent)
www.nag.co.uk	No	Fortran, Matlab	Yes	Gives literature references

Table 1

This article will discuss minimax fit. Additional articles, which will appear later this month on Embedded.com, will deal with the other cases.

MINIMAX FITTING: POLYNOMIALS AND LINES

Minimax fitting is preferred to regression when the goal is to find a fitting line with *minimum tolerance*. This makes it appropriate for some automatic control applications.

The difference between minimax and least-squares fitting is most apparent when the deviations are not symmetrical, as shown in **Figure 1**. The regression line follows the *bulk* of the data, while the minimax line tries to "split the difference" between high and low deviations. In the case shown in Figure 1, the minimax line gives an error tolerance about 40% smaller than that achieved via regression.

LIBRARIES OF DOWNLOADABLE ALGORITHMS

At the time I started working on the problem, I was not aware of a linear minimax fitting algorithm. However, I did know of a general method for polynomial minimax fitting called the *Remez algorithm* (polynomial fitting includes linear fitting as a special case). There are numerous references to the Remez algorithm on the web (including Wikipedia) and in numerical analysis textbooks.

With a little searching, I found that there is downloadable Remez code available from various Web libraries. **Table 1** summarizes the results of my investigation. Unfortunately, all the

EE|Times Virtual Conference

Upcoming Virtual Conferences

EE|Times Virtual Conference:

Maximizing the Flexibility of FPGAs

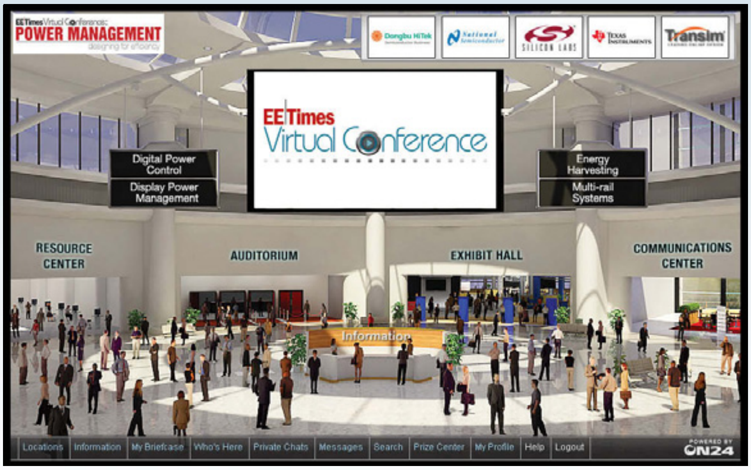
When: Thurs., June 24, 2010, 11am-6pm EST
www.eetimes.com/FPGA

On Demand Virtual Conferences

**Designing with ARM:
Engineer an Optimal ARM-based system**
www.eetimes.com/arm

**Motor Control:
Intelligent Control Maximizes Performance, Minimizes Power/Cost**
www.eetimes.com/motor

Medical Systems Design
www.eetimes.com/medical



EE Times, the leading resource for design decision makers in the electronics industry brings to you a series of Virtual Conferences. These fully interactive events incorporate online learning, active movement in and out of exhibit booths and sessions, vendor presentations and more. Because the conference is virtual you can experience it from the comfort of your own desk. So you can get right to the industry information and solutions you seek.

Why you should attend:

- Learn from top industry speakers
- Participate in educational sessions in real time
- Easy access to EE Times library of resources
- Interact with experts and vendors at the Virtual Expo Floor
- Find design solutions for your business

**For sponsorship information, please contact:
David Blaza, 415-947-6929 or david.blaza@ubm.com**

EE|Times Group

Possible configurations of maximum-error points for minimax linear fit.

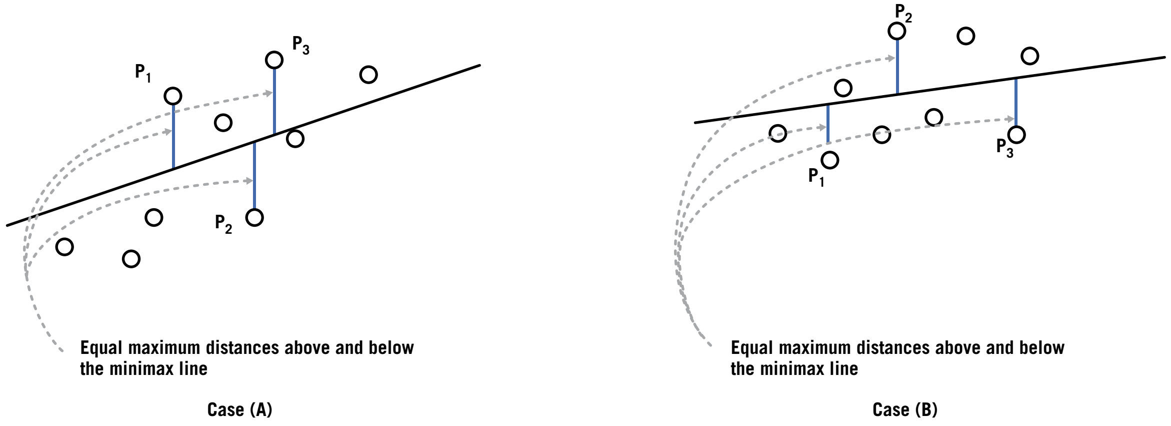

P1
P3
P2
Equal maximum distances above and below the minimax line
Case (A)
P2
P1
P3
Equal maximum distances above and below the minimax line
Case (B)

Figure 2

codes in Table 1 are designed for general purpose and were far too large and complex for the specific application I wanted.

My first thought was to try to simplify one the codes, based on the documentation provided. As shown in Table 1, only two libraries provided documentation of the algorithm, while the others gave references to the literature. I wanted to avoid digging through textbooks and journals if possible, so I started hopefully through the *www.boost.org* article on the Remez algorithm. My enthusiasm was somewhat dampened when I encountered several mathematical statements that I could easily prove were not true. My guess is that the article was written by an implementer who did not fully understand the algorithm—caveat lector!

THE HEART OF THE ALGORITHM

The Mathworks documentation on the other hand I found to be accurate and very understandable, with clear, accurate, and illustrated with excellent figures. There I found the following pearl of wisdom:

> Chebyshev … gave the criteria for a polynomial to be a minimax polynomial. Assuming that the given interval is $[a, b]$, Chebyshev's criteria states that if $P_n(x)$ is the minimax polynomial of degree n then there must be at least $(n+2)$ points in this interval at which the error function attains the absolute maximum value with alternating sign.

Now here was something I could take to the bank! In the case of a linear fit (n=1), there had to be (at least) three data points where the maximum error was achieved (here we denote the three points as P_1, P_2, and P_3 where $Pj=(x_j,y_j)$ and $x_1<x_2<x_3$). Furthermore, there were only two possibilities for these three points:

Properties (1–3) for Cases A & B.

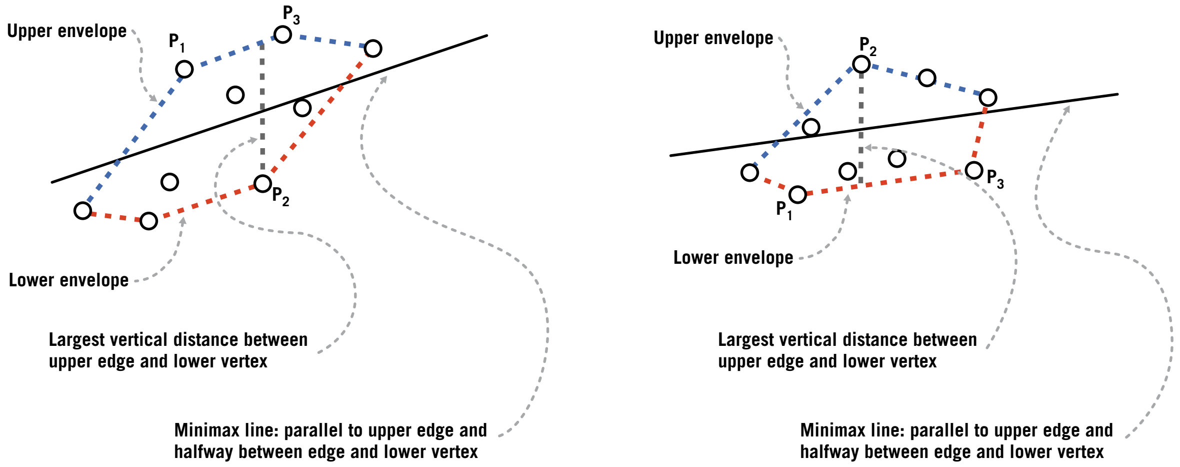

Upper envelope
P1
P3
P2
Lower envelope
Largest vertical distance between upper edge and lower vertex
Minimax line: parallel to upper edge and halfway between edge and lower vertex
Upper envelope
P2
P1
P3
Lower envelope
Largest vertical distance between upper edge and lower vertex
Minimax line: parallel to upper edge and halfway between edge and lower vertex

Figure 3

(A) P_1, P_2, and P_3 lie above, below, and above the fitted line, respectively;

(B) P_1, P_2, and P_3 lie below, above, and below the fitted line, respectively;

These two possibilities are shown in **Figure 2**. Thinking geometrically about these two possibilities quickly led to several conclusions (shown in **Figure 3**):

- P_1 and P_3 must be consecutive points in the convex hull[1] of all data points. This must be true because all data points either lie below the line P_1P_3—as in case (A)—or above the line P_1P_3, as in case (B).
- P_2 must also be a vertex point on the convex hull of all data points. This holds because all data points lie either above (case A) or below (case B) the line through P_2 that is parallel to P_1P_3.
- Segment P_1P_3 must be parallel to the fitted line. This is necessarily true because P_1 and P_3 have the same error, hence the same vertical distance to the line.

These findings prompted the following straightforward prescription for an algorithm:

1. Find all vertices of the convex hull of all data points. The vertices are divided into two sets: upper envelope (U) and lower envelope (L).
2. For every pair of consecutive vertices in U (denote these vertices as (u1,v1) and (u2,v2)):
   2.1. For all vertices (x,y) in L with u1≤x ≤ u2, find the vertex that has the largest vertical distance to the upper convex hull. This vertical distance is given by:

$$D = x\times(v2 - v1)/(u2 - u1) + (v1\times u2 - v2\times u1) / (u2 - u1) - y$$

3. Select the upper vertices $(u1,v1)^+$, $(u2,v2)^+$ and lower vertex $(x,y)^+$ that corresponds to the overall largest vertical distance D^+ found in 2.
4. Repeat steps 2 and 3, switching the role of upper envelope (U) and lower envelope (L). Select the lower vertices $(u1,v1)^-$, $(u2,v2)^-$ and upper vertex $(x,y)^-$ that corresponds to the overall largest vertical distance D^-.

5a. If $D^+ \geq D^-$, then the minimax line is parallel to the segment $[(u1,v1)^+, (u2,v2)^+]$ and lies a vertical distance $D^+/2$ below the segment;

5b. If $D^+ < D^-$, then the minimax

Listing 1 Minimax fitting.

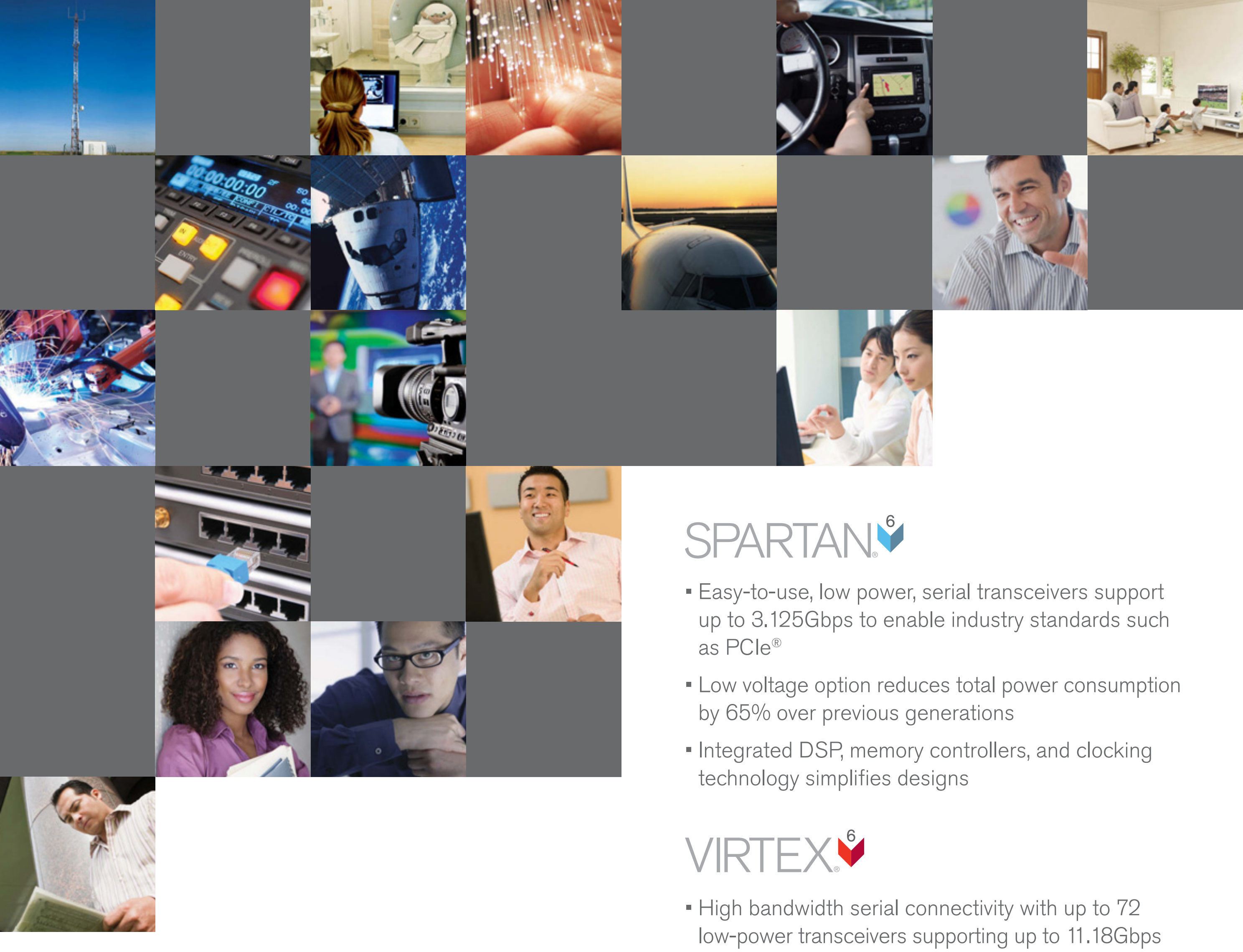
```
# ===============================s=====================================
#Chris Thron, 18 March 2010
#Input to function is the vector of Y-coordinates
#X-coordinates are assumed to be 1,2,3,....  (user can rescale at will)
#Function returns [slope, intercept] of fitted line
# ====================================================================
#
function Line_params = Minimax_line1(Y)
  #1. find upper and lower envelopes using Graham scan (reference: Wikipedia)

  #Lower and upper envelope begin at the same point
  Up = 1:2;
  Lo = Up;
  #Compute Upper envelope via Graham scan
  for ii=3:1:length(Y)
    while (Y(ii)-Y(Up(end)))*(Up(end)-Up(end-1)) >= (Y(Up(end))-Y(Up(end-1)))*(ii-Up(end))
      Up = Up(1:end-1);
      if (length(Up) == 1) break; endif
    end
    Up=[Up ii];
  end
    #Compute Lower envelope via Graham scan
  for ii=3:1:length(Y)
    while (Y(ii)-Y(Lo(end)))*(Lo(end)-Lo(end-1)) <= (Y(Lo(end))-Y(Lo(end-1)))*(ii-Lo(end))
      Lo = Lo(1:end-1);
      if (length(Lo) == 1) break; endif
    end
    Lo = [Lo ii];
  end

  #2. Find max. Lo distance between all consecutive pairs of Up points
  jj = 2;
  max_dist = 0;
  max_up_index = 0;
  max_lo_index = 0;
  for ii=2:1:length(Up)
    #2.1 For given pair of Up points, find the Lo point between with largest distance
    a_temp = (Y(Up(ii))-Y(Up(ii-1)))/(Up(ii)-Up(ii-1));
    b_temp = (Y(Up(ii-1))*Up(ii)-Y(Up(ii))*Up(ii-1))/(Up(ii)-Up(ii-1));
    while Lo(jj)<Up(ii)
      temp_dist=Lo(jj)*a_temp + b_temp - Y(Lo(jj));
      if (temp_dist > max_dist)
        max_up_index = ii-1;
        max_lo_index = jj;
        max_dist = temp_dist;
      endif
      jj=jj+1;
    end
  end

  #4, 4.1 Similarly, find max Up distance between all consecutive pairs of Lo points
  jj=2;
  max_dist2 = 0;
  max_up_index2 = 0;
  max_lo_index2 = 0;
  for ii=2:1:length(Lo)
    a_temp = (Y(Lo(ii))-Y(Lo(ii-1)))/(Lo(ii)-Lo(ii-1));
    b_temp = (Y(Lo(ii-1))*Lo(ii)-Y(Lo(ii))*Lo(ii-1))/(Lo(ii)-Lo(ii-1));
    while Up(jj)<Lo(ii)
      temp_dist=  Y(Up(jj))- Up(jj)*a_temp - b_temp;
      if (temp_dist > max_dist2)
        max_lo_index2 = ii-1;
        max_up_index2 = jj;
        max_dist2 = temp_dist;
      end
      jj=jj+1;
    end
  end
```

CONTINUED ON PAGE 32



SPARTAN® 6

- Easy-to-use, low power, serial transceivers support up to 3.125Gbps to enable industry standards such as PCIe®
- Low voltage option reduces total power consumption by 65% over previous generations
- Integrated DSP, memory controllers, and clocking technology simplifies designs

VIRTEX® 6

- High bandwidth serial connectivity with up to 72 low-power transceivers supporting up to 11.18Gbps
- Ultra high-performance DSP using up to 2016 low-power, performance-optimized DSP slices
- Integrated high-performance ExpressFabric™ technology running at 600 MHz clocking and performance-tuned IP blocks
- Proven cost-reduction with EasyPath™-6 FPGAs

Potential. Realized.

Unleash the full potential of your product design with Xilinx® Virtex®-6 and Spartan®-6 FPGA families — the programmable foundation for Targeted Design Platforms.

- Reduce system costs by up to 60%
- Lower power by 65%
- Shrink development time by 50%

Realize your potential. Visit www.xilinx.com/6.

XILINX®

CONTINUED FROM PAGE 30

```
  #5a, 5b. Choose overall maximum distance, return line parameters accordingly
  if (max_dist > max_dist2)
    slope = ( Y(Up(max_up_index+1))-Y(Up(max_up_index)) )/(Up(max_up_index+1)-
Up(max_up_index));
    Point = [Lo(max_lo_index), Y(Lo(max_lo_index)) + max_dist/2];
    intercept = Point(2) - slope*Point(1);
  else
    slope = ( Y(Lo(max_lo_index2+1))-Y(Lo(max_lo_index2)) )/(Lo(max_lo_index2+1)-
Lo(max_lo_index2));
    Point = [Up(max_up_index2), Y(Up(max_up_index2)) - max_dist2/2];
    intercept = Point(2) - slope*Point(1);
  end

  Line_params = [slope intercept];

  endfunction
```

line is parallel to the segment $[(u1,v1)^-, (u2,v2)^-]$ and lies a vertical distance $D^-/2$ above the segment;

The key remaining task was to obtain an algorithm for finding the convex hull. A Google search on "convex hull algorithm" quickly led to the "Graham scan" article on Wikipedia, which even contains a pseudocode. In the case at hand, the algorithm was made even easier by virtue of the fact that the data was sorted in order of increasing x-coordinate, and the x-coordinates were evenly spaced.

POSTSCRIPT: MORE THAN ONE WAY TO SKIN A CAT …

As I was researching for this article, I discovered a similar (but not identical) algorithm in the technical literature [Rey and Ward, 1987].

Prototype code

The prototype code in **Listing 1** is written in Octave (an open-source Matlab-compatible language). Caveat: No serious attempt has been made to optimize the code either for reduced code size or faster run-time.

Part 2 and 3 of this article will be online on Embedded.com in the near future. ■

Chris Thron is currently assistant professor of mathematics at Texas A&M University Central Texas, and does consulting in algorithm design, numerical analysis, system analysis and simulation, and statistical analysis. Previously Chris was with Freescale Semiconductor, doing R&D in cellular baseband processing, amplifier predistortion, internet security, and semiconductor device performance. He has six U.S. patents granted plus three published applications. His web page is *www.tarleton.edu/faculty/thron/*, and he can be reached at thron@tarleton.edu.

ENDNOTES:

1. The *convex hull* of a set S is the smallest convex set containing S. Alternatively, it is the intersection of all half-planes that contain S.

REFERENCES:

boost C++ libraries resources for minimax polynomial fitting:

- Minimax Approximations and the Remez Algorithm. Available from *www.boost.org/doc/libs/1_36_0/libs/math/doc/sf_and_dist/html/math_toolkit/toolkit/internals2/minimax.html*; accessed 18 March 2010 (describes code)
- The Remez Method. Available from *www.boost.org/doc/libs/1_36_0/libs/math/doc/sf_and_dist/html/math_toolkit/backgrounders/remez.html*; accessed 18 March 2010 (describes Remez algorithm)

MathworksTM resources for minimax polynomial fitting:

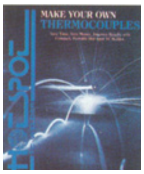
- Tawfik, Sherif. Remez Algorithm, available for download at *www.mathworks.com/matlabcentral/fileexchange/8094*; accessed 18 March 2010. (includes Matlab® code and .pdf documentation)

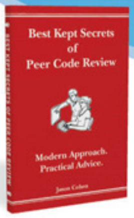
NAG library resources for minimax polynomial fitting:

- NAG Fortran Library Routine Document, E02ACF. Available from *www.nag.co.uk/numeric/Fl/manual/pdf/E02/e02acf.pdf*; accessed 18 March 2010.
- NAG Toolbox for Matlab e02ac. Available from *www.nag.com/numeric/MB/manual_22_1/pdf/E02/e02ac.pdf;* accessed 18 March 2010.

Netlib library resources for minimax polynomial fitting:

- ACM Algorithm #414, "Chebyshev Approximation of Continuous Functions by a Chebyshev System of Functions." Available for download from *www.netlib.org/toms/414;* accessed 18 March 2010.
- Golub, G.H., Smith L.H. Chebyshev Approximation of Continuous Functions by a Chebyshev System of Functions. Available from *ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/67/72/CS-TR-67-72.pdf* (This report documents the Netlib Algol code)
- Rey, C. and Ward, R. On Determining The On-Line Minimax Linear Fit To A Discrete Point Set In The Plane. Information Processing Letters 24 (1987), 97-101.





Programmer's toolbox

from page 14

fast. Not microseconds, but nanoseconds. If you don't want to use callable functions, fine, but you can't use run time speed as an excuse.

TEST, TEST, TEST

Now we get to what I consider to be the foundation of my programming style. I test. I test everything, A lot.

The reason may sound odd: I test because I hate testing. Most programmers do, but in my case it's personal. I hate testing because I hate the notion that I wasn't perfect to begin with. So I hate testing. But if I hate testing, I hate testing *later* even worse. I test every line I write, right after I write it, so I won't have to come back in shame, later, to put it right.

In the best tradition of programmed programming, I don't want to wait until days or weeks later, to find out something's wrong. I *sure* don't want to wait until the entire program is complete, like as those poor saps on the Unix system had to do. I want to keep that instant gratification going. I want to preserve the expectation of success.

That's why I test as I write. Add a few lines of code (three or four), then test.

A few consequences fall out of this approach. First, the term "code and unit test" implies that you've got a test driver for every unit. And indeed I do. No matter how small the unit under test (UUT), it gets its own test driver. And yes, I absolutely did test `iscomma` (Hey, it didn't take too long. There are only two outcomes, after all).

I'm also a big believer in single-stepping through my code. My pal Jim Adams disagrees with me on this. He says that if you're going to use single-stepping at all, you should only do it once. After all, he argues, the instruction is going to do the same thing each time.

Maybe so, but I do it anyway. Consider it part of the programmed programming paradigm. I never turn down an opportunity to get warm fuzzies. Single-stepping reminds me in no uncertain terms that my processes—mental and computerwise—are still working.

Finally, there's an implicit assumption that you have a fast IDE that can give you test results in a hurry. I've been spoiled on this point ever since I fell in love with Turbo Pascal. The idea that I could get a compile as fast as I could get my finger off the 'R' button, blew me away. And spoiled me for life.

This wasn't a problem for the Unix team. The reason the managers imposed that ridiculous "no compiling" rule was, the compiler was slow, taking some four hours per build. The team should have—and could have—figured out a way to compile and unit-test the individual modules, but they didn't. For one reason, their code was not modular; everything depended on everything else. So you couldn't just lift out one function for unit testing, you had to do the whole megillah. The programmers didn't mind though. They could launch a build, then spend the next four hours playing Unix computer games.

I once worked on an assembly-language program, an embedded program. After I'd given the team leader some software, I told him I needed to make a change to it. He said, "Well, you can't do that now. You'll have to wait for the next time we do a build." I asked when that would be. He said, "Two or three weeks."

I go, "Three *weeks*? Are you kidding me? I'm used to getting new builds in three *seconds*!" Then and there, I resolved to do my unit testing on my own test drivers and save integration until the very last thing. The idea worked like a charm. During integration, we found only one error, requiring a change in one single byte.

Expect success, I say. Get warm fuzzies often. And don't forget to test. ■

By Jack G. Ganssle

After 500,000 words

Twelve years of Catholic education taught me to hate writing. The nuns and Jesuits were very demanding and tolerated neither spelling errors nor grammatical mistakes. Getting a five page paper done seemed to require Herculean longhand efforts. My poor mother—an English major—typed a lot of these papers for her five kids on an ancient manual Olivetti.

As an engineer I found that it was happily easy to get buried in a project and respond to non-techies with grunts and scribbled schematics. They'd go away pretty quickly. But over time I took more and more pleasure in getting the comments right, with enough narrative to ensure one could completely understand the software without referencing the code itself. Annotating schematics was even more interesting as space limitations meant one had to adopt an astonishingly concise style while conveying lots of information. Then I learned that one very effective way to elicit a project's requirements was to write the user's manual first… and that a truly well-written manual was a joy the customer and a source of pride to the author. So it was a natural progression to learning to love the art of writing in other forms as well.

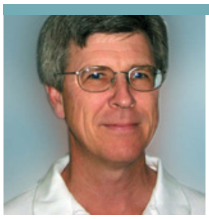
I cranked out some technical articles for a few publications, including two for *Embedded Systems Programming*, this magazine's original name, working with the magazine's most colorful editor, Tyler Sperry. Then he called and asked for a monthly column. The first installation of Breakpoints appeared June, 1990, 20 years ago this month.

The first installation of Breakpoints appeared June 1990, 20 years ago this month.

Those two decades have passed at a frightening rate. Then: mid thirties, a baby and one on the way, building a business, new house, and always a crisis at hand. Now I find myself with a very different perspective: genetically irrelevant, watching a new generation of parents starting out, enjoying friendships forged over many decades, and less preoccupied with the exigencies that always turn out to be so unimportant once a little time passes. Those babies are grown, the business sold long ago, and crises rare.

Everything changes.

And so much has changed in this industry. I don't have a copy of the June, 1990 issue of this magazine, but the very first ESP issued about a year earlier had, by rough count, ads from 60 different companies. Only a dozen or so of those outfits are still around, or selling the same sorts of products. Do you old-timers remember Ready Systems? Huntsville Microsystems? Softaid, AD 2500, Whitesmiths? All gone, sold, their products largely forgotten. Strangely, Wind River didn't advertise in that issue. But they sure bought a lot of others who did during the stock bubble of the early 2000s. In one acquisition spree ISI, after buying



Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.

Diab and SDS, was in turn purchased by Wind River, all in a matter of weeks. One friend worked for all of those companies without ever changing jobs. Intel, which now owns Wind River, had a full page ad for an 8051 in-circuit emulator.

Other companies have been born in the intervening decades. Some with the biggest ads in recent ESDs didn't exist in 1990, like Express Logic and plugcomputer (Marvell).

Electronics is the home of the fastest rate of change of about any discipline, so it's not surprising that so much we take for granted today didn't exist two decades ago. Something like 70% of all 32-bit processors are ARMs, yet that processor didn't exist as we know it. The market trends for CPUs are completely different now, too. Then, there were a plethora of different architectures, with more released on what seemed a daily basis. Today, the 32-bit trend is unifying behind the ARM line, and I suspect that as time moves on ARM's market share in high-performance embedded systems will be indistinguishable from Intel's on the desktop.

What's even more remarkable is the 8051-ication of the ARM. TI/Luminary, ST, and others have taken the MCU philosophy of massive part proliferation to the 32-bit world. It's ironic that the latest generation of these parts in their Cortex M0 incarnation sport about the same number of instructions as the very first microprocessor, the 8008. Few would have predicted that a 32-bit MCU would cost the $0.65 NXP charges for their LPC1100.

Even some technology words have disappeared. "Submicron," for instance, was the almost inconceivable realm where transistor geometries were smaller than, gasp, one micron. Today 0.045 micron is not uncommon, and 0.022 is on the horizon. Back then I was astonished at the notion of features just a single micron long, and I remain even more so now with gate oxide thicknesses just five atoms thick. What an amazing world we engineers have built!

There was no Linux and any variant of Unix was only rarely found in an embedded system. Windows CE didn't exist. GUIs of a sort were sometimes found in the embedded world, powered by home-grown libraries or a few commercial products. But a high-end embedded CPU then was a fast (like 20 MHz) 16 bitter so processor cycles were in short supply.

in January 1991, I made some predictions. Let's see how they fared: I thought C++ would win. Wrong. And I thought cubicles would disappear.

PREDICTIONS

Bill Vaughn noted: "The groundhog is like most other prophets: it delivers is predictions and then disappears." Well, in January 1991, I made some predictions in this column. Let's see how they fared.

I thought C++ would win. Wrong, at least so far. Its market share in firmware struggles along at about 30%. This is still, unfortunately, the world of C. I like C—it's fun, it's expressive, and one can do anything with it. Which is the problem. C worked well when programs were small, but it scales poorly as line counts explode.

The siren call of reuse seduced me, and I wrote that we'd find vendors selling chunks of code just as they push ICs. Databooks would list lots of firmware modules, completely spec'ed out. We'd just string them together and build the system out of components. Strike two. And what the heck is a databook? When was the last time you saw one of those?

I predicted that integrated development environments (IDEs) would track function names and tell us what their parameters are. So there's a hit as many development environments provide plenty of information about functions, variables, and program structure. But I thought that these new help resources would aid in managing a plethora of files whose function was hidden by the eight-character file name limitation of DOS and Windows, never anticipating that we'd even be able to embed spaces in names today.

It's hard to believe, but I figured management would realize how expensive we are and backstop us with lower-paid paraprogrammers and clerical help to offload all nonengineering functions from us. And I thought productivity-killing cubicles would disappear.

Perhaps the ultimate irony is that I never imagined that in-circuit emulators would nearly fade away, replaced by BDMs and JTAG. At the time my company made ICEs. Remember Keuffel and Esser, the slide rule company? In 1967 they predicted all sorts of fantastic changes, like domed cities, to come over the next century. They missed the demise of the slide rule just a few years later.

Predictions are hard. Even the Psychic Friends Network didn't see their 1998 bankruptcy in the tea leaves.

LESSONS LEARNED

To me the most interesting and compelling aspect of any endeavor is learning. After nearly a half-century of sailing, for instance, I still learn new aspects of the sport every year. In electronics and firmware things change at a much more dizzying rate than in the ancient world of rope and sails. Not a day goes by that one doesn't pick up something new.

The same goes for writing. Grammar and style are important so I've learned to keep English reference books at hand. And I've learned that the *Chicago Manual of Style* is not only useful—it's a pretty darn interesting read in its own right, at least for those who love language.

I've learned that no matter how

carefully one prunes all hint of politics and religion from a column, at times a vocal few will get outraged for a perceived right/left wing bias. I've been accused of being a miserable conservative shill and a raging left-wing socialist. But as America's political dialog has been getting more charged and less reasoned, and as the Internet has eroded all sense of manners, responses from ESD's readers are nearly always well-thought-out and polite.

I've learned how smart—and at times funny—ESD readers can be. Over two decades of writing for this magazine I've exchanged over 100,000 e-mails with you, some of whom have become good friends.

You are getting more diverse. I don't know the magazine's demographics, but e-mail increasingly comes from areas of the world where engineering was nearly unknown 20 years ago. Occasionally that correspondence is framed more aggressively than one expects in the West: "Required urgently: schematics and source code to build a system that . . . " What I haven't learned, but want to, is if this is a result of some intriguing difference in our cultures or perhaps a different take on etiquette.

Some of that diversity includes a wider range of languages; English is often not an ESD reader's native tongue. If I were wiser I'd adopt that essential fact and stick to simpler sentence structure and mostly monosyllabic words. If the goal is to communicate it makes sense to subset the language to one that is comprehensible to the entire readership. Rather like applying MISRA to C or SPARK to Ada.

But that's no fun. English is marvelously rich in expressive words and constructs. It's delightful to surf the language.

I've learned the peril of idioms, which will trip up a conversation even between an American and a Englishman. Add those to a publication read by folks from a hundred more distant cultures and you're sure to "come a cropper." For instance, I recently received this e-mail:

> *Dear Professor,*
>
> *Recently, I have read your article. The last section of the article: "At the risk of sounding like a new-age romantic, someone working in aroma therapy rather than pushing bits around, we've got to learn to deal with human nature in the design process. Most managers would trade their firstborn for an army of Vulcan programmers, but until the Vulcan economy collapses ("emotionless programmer, will work for peanuts and logical discourse") we'll have to find ways to efficiently use humans, with all of their limitations."*
>
> *I have the following questions: "Working in aroma therapy rather than pushing bits around"—what do the both mean, and what the writer want to express?*
>
> *"Trade their firstborn for an army of Vulcan programmers"—Here I think "firstborn" maybe his (her) employees, but what Vulcan programmers stand for? I have little knowledge about Roman myth.*
>
> *"Vulcan economy collapses"—Vulcan economy stands for?*
>
> *"Work for peanuts and logical discourse"—work for peanuts and logical discourse? I can't find any clue of a relation between them, so I really don't know what the sentence mean.*
>
> *I have searched a lot of Web pages about these, but failed to get considerable answers.*
>
> *sincerely long for and thank you for your help!*

I've learned how smart—and at times funny—ESD readers can be. Over the two decades of writing for this magazine I've exchanged over 100,000 e-mails with you, some of whom have become good friends. My e-mail inbox is always swamped so it takes days for me to reply, but the questions, comments and critiques are so perceptive and interesting I try to respond to every one.

This discourse with you, gentle readers, is the best part of writing a column. As Dean Martin used to say: keep those cards and letters coming! And thanks for the dialog. ■

ebook Reader
ebook Demos

R&D Prototype
PCB Assembly
$50 in 3-Days

Advanced Assembly specializes in fast assembly for R&D prototypes, NPI, and low-volume orders. We machine-place all SMT parts and carefully follow each board through the entire process to deliver accurately assembled boards in three days or less.

R&D Assembly Pricing Matrix/Free tooling and programming

Up to # SMT Parts	25	50	100	150	200	250	300	Over 300
1st board	$50	$85	$105	$155	$205	$255	$305	Call for Pricing
2nd board	$30	$55	$65	$95	$125	$165	$185	
Each additional board	$25	$35	$45	$65	$95	$125	$155	
Stencil	$50	$50	$50	$50	$50	$50	$50	

aapcb.com/esd5
1.800.838.5650

Advanced Assembly®
SMT ASSEMBLY FOR ENGINEERS℠

Lead Free

SO 9001:2008 Certified

The new standard for pcb assembly